University of Ljubljana
Faculty of Computer and Information Science

Črt Kristl

Genetic Approach to Strategic Games

graduate thesis

professional study programme of computer and information science

Mentor: izr. prof. dr. Gašper Fijavž
Co-Mentor: doc. dr. Andrej Brodnik

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Črt Kristl

Genetski pristop k strateškim igram

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Gašper Fijavž
Somentor: doc. dr. Andrej Brodnik

Ljubljana, 2016

The Faculty of Computer and Information Science issues the following thesis:

Topic of the thesis:

Action heroes in strategic computer games may possess different abilities with respect to their inherent properties and the state of the game. Construct a system for comparing abilities of heroes in game Warcraft III. Include also a system for optimizing their abilities based on evolution algorithms.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Akcijski junaki v strateških računalniških igrah imajo lahko različne sposobnosti glede na njihove inherentne lastnosti in trenutni potek igre. Na primeru igre Warcraft III izdelajte način za primerjavo junakov, ki naj vključuje genetsko optimizacijo njihovih lastnosti in sposobnosti.

Declaration of the diploma work authorship

Undersigned, Črt Kristl, author of the graduate thesis with a topic:

Genetic Approach to Strategic Games

I confirm, that:

- the graduate thesis was made on my own, mentored by izr. prof. dr. Gašper Fijavž and co-mentored by doc. dr. Andrej Brodnik,
- the electronic copy, the title, the abstract of the graduate thesis are identical with the printed copy of the graduate thesis,
- I agree with publishing copies of the graduate thesis on the Internet through the university web archive.

Ljubljana, January 20, 2016 Signature of the author:

Contents

Commonly Used Abbreviations

abbreviation	meaning
AI	artificial intelligence
API	application program interface
CPU	central processing unit
DPET	damage per execution time
DPS	damage per second
EDPS	effective damage per second
FPS	frames per second
HP	health point(s)

Abstract

Title: Genetic Approach to Strategic Games

In this thesis we develop and implement a genetic algorithm to optimize a set of talents, equipment and sub-attributes of characters in the game Warcraft III and its modification The Kingdom of Kaliron. Finding the optimal set where a character performs the best in fights against enemies is a combinatorial problem for which we use a genetic algorithm to solve.

To be able to evaluate a character, we implemented a simulation that required deep knowledge of game mechanics and programming principles of Warcraft III. We also used reverse engineering as a tool.

We ensured convergence of a genetic algorithm with the use of population islands, which are disjoint subpopulations with weak mutual interactions, and with careful choosing of genetic algorithm parameters. We also implemented genetic algorithm memory, which helps create better initial individuals when creating new populations. Finally, we used parallelization to reduce the running time of the algorithm.

Keywords: genetic algorithm, computer games, optimization, parallelization, simulation.

Povzetek

Naslov: Genetski pristop k strateškim igram

V delu razvijemo in implementiramo genetski algoritem s katerim optimiziramo nabor znanj, opreme in lastnosti akcijskih junakov v strateški igri Warcraft III, oziroma njeni različici The Kingdom of Kaliron. Izbiro optimalnega nabora, takega pri katerem je akcijski junak kar se da uspešen v bojevanju z nasprotniki, zapišemo kot problem kombinatorične optimizacije, za njegovo reševanje pa uporabimo pristop z genetskim algoritmom.

Za oceno sposobnosti junaka smo implementirali simulacijo, za katero smo potrebovali natančno poznavanje mehaničnih in programskih principov igre Warcraft III. Med drugim smo uporabljali metode vzvratnega inženirstva.

Konvergenco genetskega algoritma smo zagotovili z uporabo otokov, ločenih podpopulacij s šibko medsebojno interakcijo, in s pazljivo izbiro parametrov genetskega algoritma. Poleg tega smo genetskemu algoritmu dodali spomin, ki pri ustvarjanju novih populacij pripomore k boljšem začetnem stanju osebkov. Ustrezno časovno učinkovitost pa smo pridelali s paralelizacijo metode.

Ključne besede: genetski algoritem, računalniške igre, optimizacija, paralelizacija, simulacija.

Razširjeni povzetek

V delu se ukvarjamo z iskanjem najboljših kombinacij parametrov akcijskih junakov v igri Warcraft III, oziroma njeni različici The Kingdom of Kaliron. Akcijske junake imenujemo *heroji*, ki imajo različne sposobnosti in lastnosti. Kriterij za najboljšega heroja smo definirali glede na škodo, ki jo povzroči v boju z nasprotniki v določenem času.

Na splošno je v večigralskih računalniških igrah zelo pomembno, da so elementi igre v ravnovesju. To pomeni, da noben element ni boljši ali slabši od drugega, čeprav imajo lahko drugačne lastnosti in sposobnosti. V našem primeru je to še bolj pomembno, saj slabši heroji ne bi bili izbrani, in bi bil čas vložen v razvoj takih herojev odveč. Program, ki smo ga razvili, se lahko uporabi za primerjanje herojev in spreminjanje lastnosti, še preden so vključeni v igro.

Uvod

V igri Warcraft III — The Kingdom of Kaliron je na voljo več tipov herojev. Vsak heroj ima svoj nabor znanj, atributov, in opreme. Nabor znanj je prikazan kot linearno drevo, v katerem ima vsako vozlišče zapisano število dodeljenih točk. Atributi so polja številk, medtem ko je oprema definirana kot vsebina dvanajstih polj, v katerih imamo različne tipe opreme. Poiskati želimo takšno kombinacijo znanj, atributov in opreme za določenega heroja, ki povzroči največ škode na sekundo.

Tak problem lahko rešujemo na dva načina: s stohastičnimi in z deter-

mističnimi metodami. Razlika med načinoma je, da stohastične metode ne zagotovijo natančnega rezultata, pri deterministčnih metodah pa dobimo pravilen rezultat, ampak potrebujejo več procesorske moči in posledično več časa, saj morajo preveriti vse možne kombinacije. Ker so vrednosti med seboj odvisne, problem hitro postane nerešljiv z deterministčnimi metodami.

Za način iskanja najboljše kombinacije nabora parametrov smo izbrali genetski algoritem, eno od stohastičnih metod, saj nam omogoči, da lahko pridemo do *optimalne* rešitve, ne da preverimo vse možne kombinacije. Genetski algoritem deluje tako, da na začetku zgradi naključno populacijo herojev z različnimi parametri, nato pa v teku kombinira in mutira te osebke. Tak osnovni genetski algoritem ni zadoščal našim potrebam, saj so bili rezultati preveč razpršeni in nezadovoljivi.

Genetski algoritem

Naš genetski algoritem je sproti dinamično spreminjal parametre, ampak samo v primeru, ko se je konvergenca ustavila ali upočasnila. Poleg tega smo spremenili koncept populacije. Namesto ene populacije smo ustvarili *otoke populacij* s šibko medsebojno interakcijo, kar je pripomoglo k boljši kvaliteti končne rešitve, saj so bile te populacije med seboj neodvisne, dokler se konvergenca ni ustavila — takrat so se začele mešati. Genetski algoritem je operiral tudi s *spominom*: v primeru slabe konvergence smo ustvarili nove populacije predvsem iz kvalitetnih osebkov, ki smo jih opazili v prejšnjih generacijah. Pri tem smo se (večinoma) uspeli izogniti ne-najboljšim lokalnim optimumom.

Pravila igre so zahtevala tudi, da heroji nimajo premalo atributov (npr. zdravja), saj ne bi preživeli v boju s težkim nasprotnikom. Te omejitve lahko uporabnik definira sam v konfiguracijskih datotekah. Če je heroj imel premalo atributov je dobil kazen, ki se je odražala negativno pri rezultatu *kriterijske funkcije*. V nasprotnem primeru je dobil bonus. Tako smo zagotovili, da genetski algoritem daje prednost tistim osebkom, ki so čim bližje

tem omejitvam ampak še vedno povzročijo največ škode.

Želeli smo doseči tudi dobro časovno učinkovitost. Zato smo implementirali preprost način povzporejanja otokov populacij. Vsaka populacija teče na svoji niti, medtem ko je ena nit odgovorna samo za izpisovanje rezultatov v predpisanih časovnih intervalih. S tem smo dosegli skoraj linearno pohitritev časa izvajanja algoritma glede na število fizičnih procesorskih jeder.

Simulacija

Kriterijska funkcija genetskega algoritma je definirana kot simulacija igranja Warcraft III. Za implementacijo simulacije je bilo potrebno natančno poznavanje mehaničnih in programskih principov igre. Nekaj teh smo ugotovili z uporabljanjem metod vzvratnega inženirstva. Opazovali smo, kako se izbrane pomnilniške celice spreminjajo kot odziv na naše akcije v igri.

Osnova simulacije je strojni igralec, ki igra heroja, ki napada "practice dummy objekt" in poskuša narediti čim več škode v naprej določenem času. V tem času strojni igralec ovrednoti svoje možnosti in izbere najboljšo.

S strojnim igralcem želimo modelirati igranje človeških igralcev. Pri tem smo želeli, da je strojni igralec po kvaliteti igranja primerljiv z najboljšimi človeškimi igralci, nikakor pa ne sme biti od njih bistveno boljši. V takem primeru bi bili naši rezultati neuporabni. Med drugimi je strojni igralec simuliral zgrešene klike (napake igralca), odzivni čas igralca na dogodke, in premikanje herojev. Potrebno je bilo tudi razviti algoritem za izbiranje najbolj učinkovitih *urokov* v določenem trenutku, ki je bil implementiran kot funkcija škode, ki bi jo povzročil urok, v odvisnosti od časa, ki bi heroj moral počakati, da bi se ta urok izvršil. Za simulacijo bolj kompleksnih bojev mora strojni igralec tudi razumeti koncept premikanja heroja po terenu.

Konfiguracija

Konfiguracija programa je shranjena v nekaj datotekah. Vsaka datoteka je namenjena svojem področju konfiguracije: konfiguracije za heroje in konfiguracija za splošne parametre kot so parametri boja in genetskega algoritma. V splošni konfiguracijski datoteki lahko poljubno spremenimo delovanje genetskega algoritma in simulacije, v konfiguraciji za heroja pa lahko spremenimo lastnosti določenega heroja ter porazdelitev atributov, v primeru ko ne želimo, da genetski algoritem sam poišče ta atribut.

Rezultati

Uspešnost našega algoritma primerjamo s tipičnim genetskim algoritmom (brez naših izboljšav). Rezultati tipičnega genetskega algoritma so bili nezadovoljivi, saj v večini primerov niso našli optimalne rešitve[1]. Tipičen genetski algoritem se je velikokrat zataknil v lokalnem maksimumu in ostal tam, saj ni imel metod za spreminjanje raziskovalnega prostora. Zaradi tega je bil tudi standardni odklon veliko večji kot pri naši izboljšavi. Večanje število osebkov v populaciji je izboljšalo kvaliteto populacije, toda ni pomagalo pri reševanju iz lokalnega maksimuma.

Z našim genetskim algoritmom smo tudi primerjali uspešnost herojev skozi potek igre. Heroji si so najbolj enakovredni pri nivoju 40, ki je tudi najvišji dosegljiv nivo v igri. Naš program omogoča simulacijo tudi višjih nivojev, toda oprema za nivoje nad 40 ni implementirana v igri.

Za konec smo primerjali še heroje na nivoju 40 z realnimi omejitvami[2]. Na tem nivoju so si heroji večinoma enakovredni, razen treh, ki so namenjeni samo povzročanju čim več škode. Drugi heroji imajo še dodatne sposobnosti, ki pomagajo ostalim herojem v skupini, med drugimi jih tudi naredijo močnejše.

[1] Optimalna rešitev je tu definirana kot najboljša rešitev, ki jo je našel naš genetski algoritem.

[2] Realne omejitve so tiste pri katerih heroj preživi v najtežjih bojih v igri.

Z rezultati smo zadovoljni, saj smo dobili dobre rešitve v sprejemljivem času. Ker je zahtevnost tega problema zelo velika, ne moremo vedeti, če je katerakoli od dobljenih rešitev najboljša možna.

Chapter 1

Introduction and Background

1.1 Warcraft III

Warcraft III: Reign of Chaos is a high fantasy real-time strategy video game released by Blizzard Entertainment in July 2002 [1]. It is played from a top-down view, where players control several units at once. It takes place on a map of different sizes with various terrain features like water, mountains, or cliffs. Players are tasked with building settlements to construct armies and gain resources. They must defend themselves against other players and ultimately conquer their enemies.

Warcraft III introduces a special unit type called *hero*. Heroes gain experience points by killing enemy units, which allow them to level-up. Progressing up a level unlocks new abilities and increases the hero's power. They can also equip items that can grant special effects and enhancements to the hero's attributes. Heroes are often very important in an army as they can wreak havoc on the battlefield if left unchecked.

Along with Warcraft III comes a "World Editor" program that allows users to create their own custom scenarios and maps. The World Editor is a powerful tool that can completely alter the game experience. Over the years people have developed many custom maps that became very popular. A few of the most popular maps got noticed by game companies who took the ideas

and created their own stand-alone games. One such example is Dota 2 [2].

This thesis is based on one of the custom maps called *The Kingdom of Kaliron*.

1.2 The Kingdom of Kaliron

The Kingdom of Kaliron is a custom map for Warcraft III that transforms the way the game is played. There is no longer any base building and you control only one main unit —- a hero. The game is played much like *World of Warcraft* from a top down view. Its main goal is to complete quests and gain experience throughout your journey and ultimately beat difficult *bosses*, which drop items for your hero. It was designed to be played in a multiplayer environment where each player would choose the hero that complements the team the most.

There are many different types of heroes, each one excelling in a different role. The three main roles are *tank*, *damage dealer* and *healer*. A tank's job is to draw attention of monsters to themselves so that they do not attack weaker heroes. A damage dealer's job is to cause as much damage as possible to prevent monsters from growing in numbers. A healer is responsible for keeping everyone alive which is often the most difficult task. An example of a fight is shown in Figure 1.1.

1.2.1 Anatomy of a Hero

A hero is a unit, see also Section 3.1.3, that automatically attacks at a specific rate at range or in melee, but can also *cast spells* and use *abilities*. Automatically attacking is often trivial compared to the damage abilities do. Each ability has an associated *mana cost* and *cooldown* before it can be used again. Abilities and normal attacks scale with attack or spell power depending on the type of the hero and are backed up by formulae. They were found by altering Warcraft III's memory and finding instructions that access those memory addresses, also called reverse engineering.

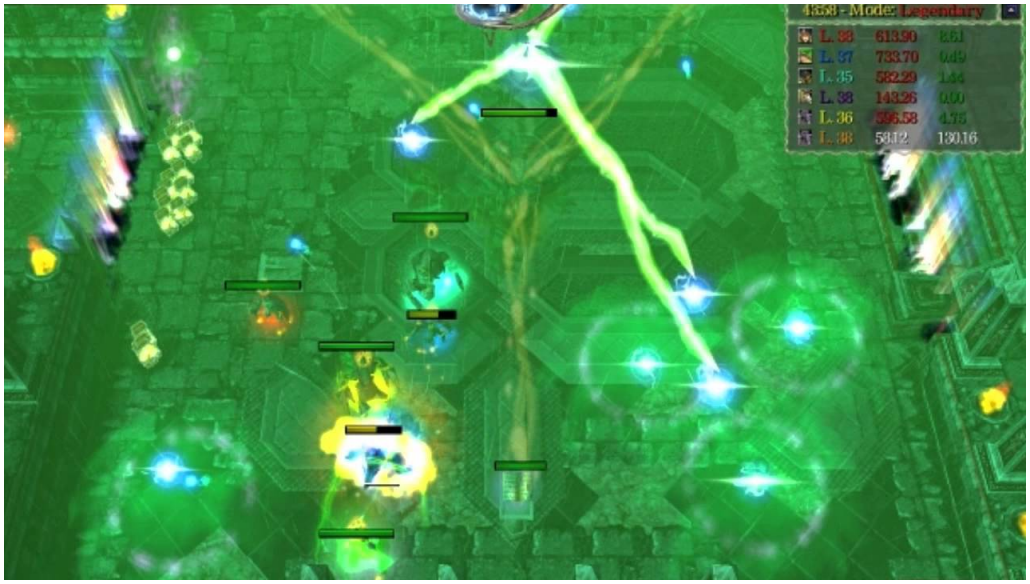

Figure 1.1: One of the last bosses. Each hero has its own role in the fight. There's a damage/heal meter in the top right (damage/heal per second).

A hero has several attributes:

- *health* — a number that must be greater than zero at all times, otherwise the hero dies,
- *mana* — a self-replenishing resource that is spent by using abilities,
- *damage* — how much damage will the hero's normal attack do,
- *armor* — when taking damage, that damage will be reduced proportionally to the amount of armor the hero has.

Beside those, there are eight other sub-attributes that the player can put points into, which influence the above statistics. Each type of hero will be based on a different sub-attribute because damage dealers may not want to put many points into *constitution*, which increases total health pool, whereas tanks will.

Each hero has a talent tree (example shown in Figure 1.2), which is a collection of skills and skill and attribute improvements. Individual talents

Figure 1.2: Talent tree of Pyromancer, a damage dealer.

in the tree are often locked until the previous talent in the chain is fully unlocked. Players are able to spend points on those talents to gain new abilities and effects. Different talents have different capacities.

A hero can wear equipment, also called items, which bolster different attributes as shown in Figure 1.3. There are a total of twelve slots for items. Items contribute to hero's power so much that without them they cannot even compare. As such, choosing the right items is an important factor in playing the game.

Every time a hero progresses to the next level in terms of experience, they get half a talent point (one for every two levels) and four sub-attribute points to spend. This along with numerous combinations of items poses a difficult problem to solve — what is the best combination of talents, items and sub-attributes for a specific task? For the purposes of our thesis we focused on finding optimal combinations of parameters for damage dealing heroes.

Figure 1.3: Inventory of a hero. Effects of the currently selected item are listed in the top right.

1.3 Combinatorial Problem

To compare heroes with each other we chose to measure their ability to cause damage to their enemies within a certain period of time. The problem was to find a combination of parameters that would output the highest *damage per second* (DPS) for the duration of a fight. With each hero getting to approximately level 40 (21 talent points and around 200 sub-attribute points divided into eight buckets and twelve slots for items) the number of possible combinations is too big to be solvable by using brute force methods. We needed to find a way to discard bad cases before they are evaluated, because the evaluation process is rather complex.

One such approach was using a genetic algorithm. By combining good combinations and getting rid of bad ones early on, the final solution would/may converge to the best possible one.

1.4 Genetic Algorithm

A genetic algorithm is a search heuristic that mimics the process of natural selection [3]. However, for it to work properly, the right set of parameters has to be chosen. If the parameters are not properly configured, the genetic algorithm will take a longer time to complete and it may not find good results [4].

The basic workflow of a genetic algorithm is shown in Figure 1.4. The whole algorithm starts by initializing a set of individual *chromosomes* that form the initial *population*. These individuals get rated according to a *fitness function*, which assigns each individual a score. Individuals with greater scores will be more likely to reproduce in the recombination process, and the ones with poor scores will get eliminated. This repeats until a satisfactory solution is found or the search is cancelled.

Genetic algorithms are very prone to getting stuck in a local maximum. To prevent this, one could increase the mutation rate, thus redirecting the search into another search space. However, in our case we found out that this often did not work because the individuals were too complex to be viable after mutating. We implemented several different techniques which will be discussed in Section 2.2.

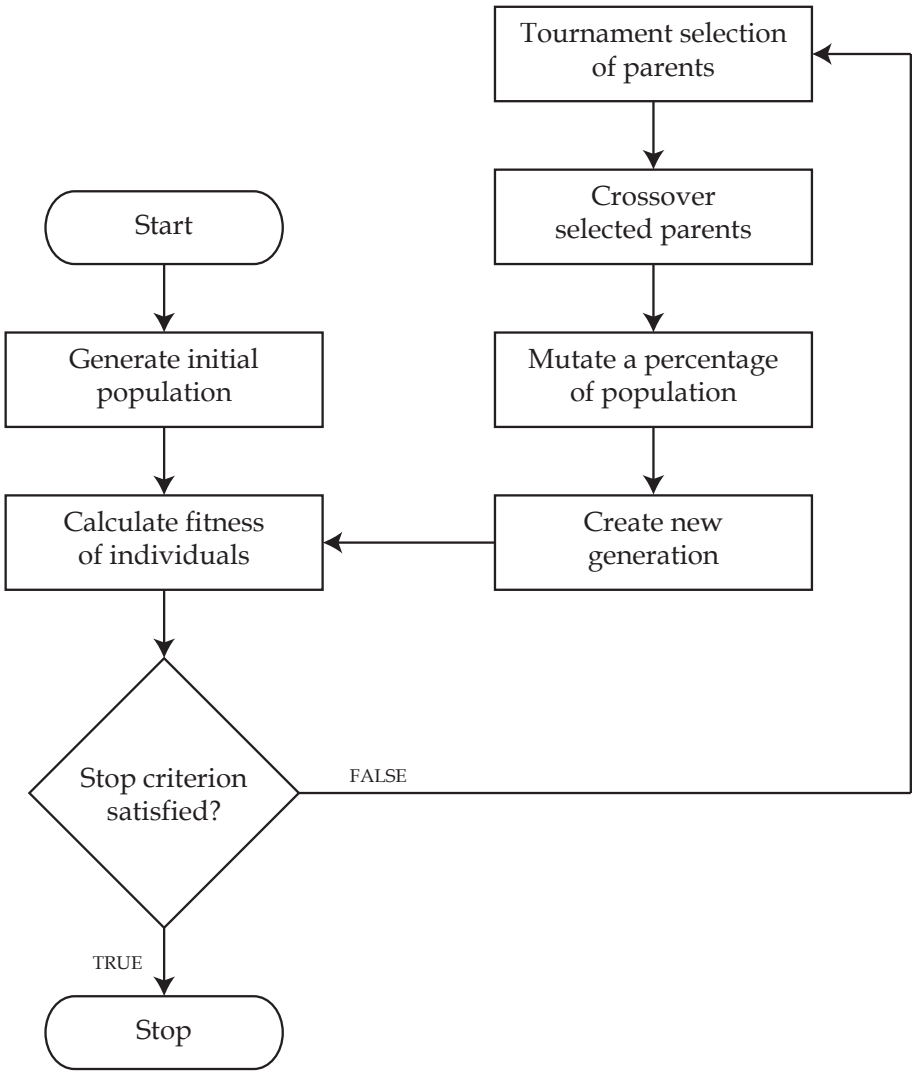

Figure 1.4: Simplified genetic algorithm flow chart.

Chapter 2

Architecture and Design

2.1 Data Mining

A custom map in Warcraft III is packaged into a file that contains all the images and scripts associated with it [5]. Among them is a JASS2 script [6] file which contains all of the game's logic. Map makers often scramble this file to protect themselves from theft.

Every ability in the game has its own function to determine the potency of its effect. There were two ways to find them; first was to read the scrambled JASS2 script file, second was to alter Warcraft III's memory and execution code by reverse engineering. The latter was much faster for our purpose. A software called Cheat Engine, which is a program that has debugging and memory reading functionalities [7], was used to accomplish that.

Along with abilities, some descriptions of attributes were also incorrectly displayed so they had to be found by reading the actual memory of the game and see how values were changed when points were assigned to them.

Items in game also had to be data mined and a database of items had to be created so that they could be easily accessed during evaluation. The only way to do that was to read the scrambled map file and parse it properly. Our program can also output the items to a text file for inspection.

2.2 Revised Genetic Algorithm

A basic genetic algorithm, as described in Figure 1.4, was not adequate for this problem because it was not possible to find a satisfactory solution under normal circumstances. It was, however, possible to obtain good solutions at the expense of increasing the size of population to 10000. But in such a case the convergence speed was too low to be practical — other means were necessary.

The purpose of this algorithm was to solve a very large number[1] of combinations on average. We had to be innovative with regards to optimizing the genetic algorithm, creating some very problem specific heuristics, which may not work in general.

2.2.1 Specimen Structure

A specimen S is comprised of three components (*chromosomes*):

- *talent tree*,
- *sub-attributes*,
- *item set*.

Each component has its own implementations for performing crossover and mutation. When performing crossover between specimens, each component might have different parents.

Talent Tree

A talent tree is represented as an array of numbers that are either 0 or 1. They are also grouped so that talents that depend on each other are in the same group. That way adding and removing points is simplified – when removing, take the first taken spot from behind and when adding, add to

[1] The total number of combinations could be $> 10^{24}$ with each taking several milliseconds to evaluate.

the first empty spot from front. This is possible because talent dependencies never fork out; they are linear.

Crossover and mutation operations then become very simple as shown in Figure 2.1.

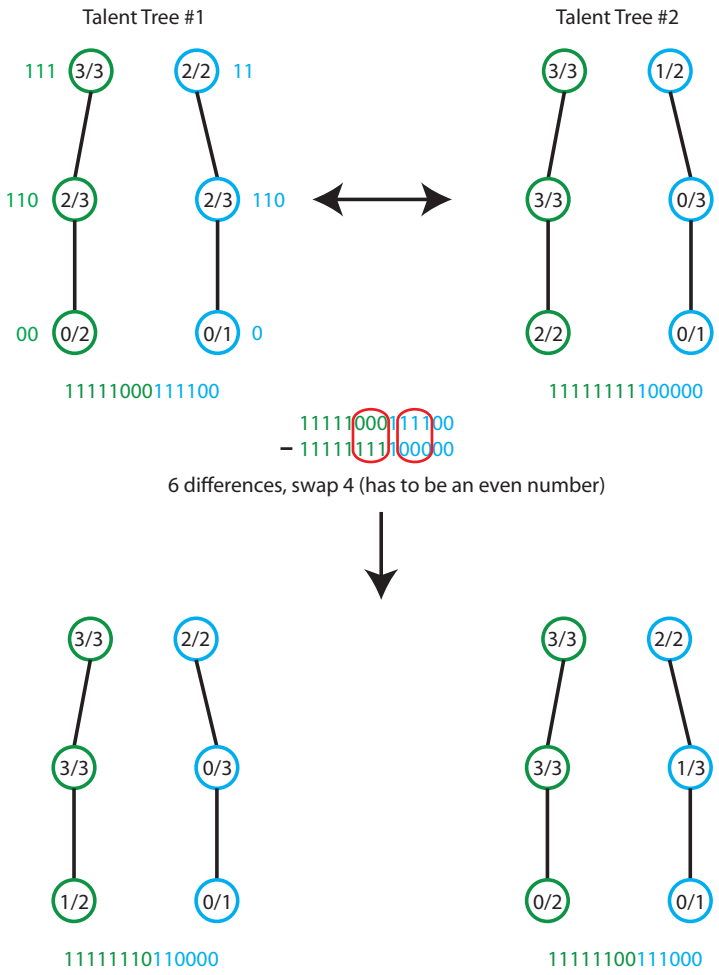

Figure 2.1: Crossover between two talent trees. Each talent chain is marked with its own color.

Algorithm 2.1 Talent Tree Crossover

Input: Parents - two talent tree chromosomes

PC - percentage of crossover

Output: Two unique talent tree chromosomes

1: *nDifferences* ← number of different bits in the input chromosomes
2: *nSwaps* ← *nDifferences* · PC
3: *children*[2] ← Parents
4: **for** $i < nSwaps$ **do**
5: SWAP(*children*[0], *children*[1])
6: **return** *children*

Swap function takes two indices in the arrays of bits, determines which group they belong to and then assigns and deducts points accordingly. Both talent trees have to end up with the same number of points or the result is invalid.

Sub-Attributes

There are eight different sub-attributes: *strength*, *constitution*, *endurance*, *reflexes*, *dexterity*, *intelligence*, *wisdom*, and *spirit*. The only requirement is that each sub-attribute must have at least one point assigned to it.

Internally they are represented as an array of numbers that are either 0 or 1. When performing crossover, two identical sub-attribute chromosomes are treated as if they have nothing in common. For example, 3|1|2 and 2|2|2 expands into 111001001100 and 000110110011 (see Figure 2.2). After that, normal crossover is performed by swapping bits randomly while keeping the total amount of points the same.

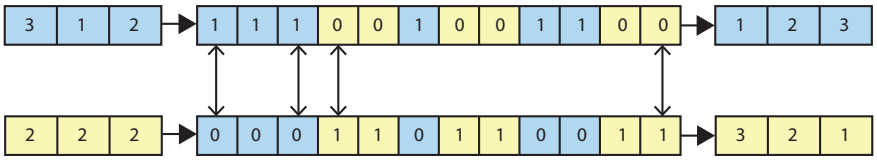

Figure 2.2: Example of a sub-attributes crossover. Left side: two chromosomes of sub-attributes. Center: internal representation during crossover. Right side: result.

Item Set

There are more than ten items per slot with twelve slots for each hero. Each item slot holds a list of possible items that can be equipped for the selected hero. When creating that list it also checks whether items can be worn by that hero (hero level, item type, ...).

When performing crossover, items are randomly swapped between specimens. They are represented as an array of numbers which correspond to their indices in the list of all the possible items for that slot.

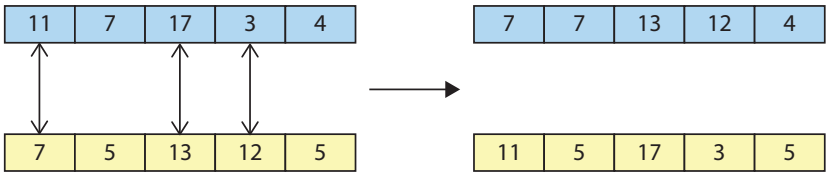

Figure 2.3: Example of an item set crossover. Numbers represent item indices.

2.2.2 Fitness Function

Fitness function $\varphi(S)$ is defined as a simulation (more about this in Section 3.2) of the game without user interaction; instead, it is played by an

artificial intelligence (AI). The AI simulates what a human player would do in the same circumstances. The output of this simulation is the value of DPS based on which penalty is calculated and subtracted from it. This results in EDPS - the quality of a specimen.

2.2.3 Evolving the Specimens

After specimens in a population get evaluated, they are selected for creating offspring based on a tournament selection. Tournament selection is a method of choosing a specimen from a population of specimens in a genetic algorithm. Tournament selection involves running several *tournaments* among a few specimens chosen at random from the population [8]. The probability for a specimen to win in a tournament is $P(S \text{ wins}) = p \cdot (1-p)^{\text{rank}(S)}$ where rank is the rank of the specimen which is based on effective DPS (EDPS) starting with 0. Winners of these tournaments are then selected to create offspring using the crossover methods. A percentage of them are also mutated to create mutated offspring, prioritizing the best specimens.

Note that a few of the fittest specimens are never deleted; this process is called *elitism* [3].

2.2.4 Metaheuristic Methods

Metaheuristics are strategies that guide the search process. The goal is to efficiently explore the search space in order to find better solutions. Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes [9]. We implemented a local search procedure for optimizing the best performing specimens.

Beside normal evolution, every n generations the algorithm optimizes top x specimens, where x is number of elites multiplied by two. This optimization permutes talent tree and item set for each specimen. It performs this several times per specimen and retests them afterwards. If the results are better, it replaces their chromosomes with the new ones.

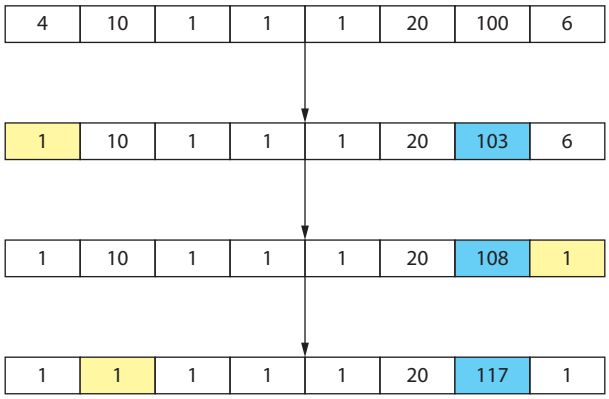

Figure 2.4: Four iterations of sub-attribute local search. This often results in large improvements. Blue and yellow colors indicate changed values from previous iteration.

Sub-attributes were often the cause of prolonged searches because there were so many combinations possible. However, heroes often perform the best if most of the points are in one sub-attribute. A local search optimization was developed that is run in two parts. In the first part all sub-attribute points that are not in the main sub-attribute are transferred to it as long as it satisfies the constraints (to be explained later in subsection 2.2.9). If the result is not better, it is reverted back to the original state. In the second part it tries to transfer everything to constitution and then through it to the main sub-attribute. This optimization (Figure 2.4) finds better solutions and sometimes even finishes the search. The only downside of this optimization is that if it is run too early, it can skew the population towards a bad local maximum. That is why the first part of this optimization is run only once every nine generations, while the second part is run once every eighteen generations.

Limited testing showed that running these methods periodically produced better quality solutions than successive runs.

2.2.5 Dynamic Adjusting

The genetic algorithm compares the best specimen in a population each generation with one from the previous generation to decide whether to adjust the parameters. The comparison function is defined as

$$|\varphi(S_p) - \varphi(S_c)| > \varphi(S_p) \cdot 0.004 \cdot \left(1 - \frac{g}{g_{max}}\right)^4 \tag{2.1}$$

where S_p is previous generation's best specimen, S_c is current generation's best specimen, g is index of current generation starting from 0 and g_{max} is the maximum number of generations to process. The constants were experimented with until we found the results agreeable.

If (2.1) is not satisfied, the convergence is considered to be halted and the algorithm will increase its internal counter by 1. If this counter goes over a specific threshold, the algorithm will begin tweaking its parameters to allow for more mutations and make the tournament selection easier to win for worse specimens. Letting worse specimens win more often coupled with elitism and more mutations enables the algorithm to examine a completely different search area. Should the best specimen be repeated further, it will discard the population and create a new one with an improved initial population based on previous results.

The reason why inequality (2.1) is defined as a function of the current generation is to promote exploration of the search space by allowing populations to be discarded more easily at the beginning. Populations will not be discarded near the end of the run of the genetic algorithm because such populations would not have enough time to mature before the genetic algorithm terminates.

2.2.6 Memory

The algorithm keeps track of the best DPS number for each parameter during its evaluation. When creating new populations, it uses this to calculate bias for parameter values and then selects the parameters in such a way that the ones with higher bias are more likely to be selected.

Bias calculation is specific to each structure — talent tree, sub-attributes and item set.

This improves the general quality of a population every time it is reset, but not so much that it necessarily would converge to the same local maximum as a result.

2.2.7 Exhaustive Testing Phase

When testing the algorithm, we often found that it gets stuck in a local maximum where only a few item flips were needed. However, item performance depended on sub-attribute distribution so once it hit a local maximum, it was very unlikely to find better item sets. This phase explores many more combinations of items that would normally be explored because we are limited by population size and speed of the algorithm.

When the algorithm is run it first runs normally until it *hits a wall* and discards the population. At this point the solution may have already been found. However, in case it has not been found, the algorithm tries to test all the possible items by inverting bias values for items for each population until all the item slots have been inverted. It picks two item slots to invert for each population (e.g. helmet and shoulder) until there are none left.

Inverse of a bias value is defined as

$$invertedBias = bias \cdot \frac{seenMax}{seen + \frac{seenMax}{200}}$$

where *seen* is the number of times this parameter value appeared in all the populations and *seenMax* is the number of times the most common item appeared. This forcefully promotes items that were not popular during evolution due to small population size or bad chance.

During this *probationary* period discarding population is much faster because the top specimen comparison is weak (i.e. 996 DPS compares favorably to 1000 DPS) and the number of required repetitions of top specimens is reduced drastically. Item permutations are also disabled so that they do not interfere with item exploration.

2.2.8 Population Islands

In the beginning of the algorithm, we only had one population to work with. This population tended to get stuck in a local maximum and never recover from it due to heavy inter-dependency between the chromosomes. Due to that, we ran several instances of the algorithm at once only to observe that each instance hit a different local maximum (potentially found the solution as well). We figured that having more populations run at once by default would improve the quality of the solutions.

The algorithm creates a number of disjoint populations equal to the amount of logical cores in the processor, with a minimum of four. When these population islands start stagnating, they request specimens from other islands (10 percent of the population, usually taking the top specimens). This helps avoiding local maxima in the population and improves diversity.

2.2.9 Constraints and Penalties

The most basic constraints are ones that are imposed by the game itself, such as number of talent points available for each level, wearable items and so on. Those constraints are enforced before the program even begins and will quit if the user somehow entered invalid data into the configuration file.

We often want to create a character that has enough health and other attributes so that it does not die to enemies, yet still performs as well as possible. It is possible to specify target values for those attributes in the configuration files for heroes. When evaluating fitness function, if those values deviate from targeted values, that specimen's score is then adjusted either negatively or positively, depending on the attribute. Without this functionality we often got results that had the highest DPS, however, the characters were not playable due to extremely low attributes that were mandatory (e.g. health).

There are three user-defined constraints: minimum health points (HP), minimum movement speed and minimum HP regeneration. EDPS is then

penalty subtracted from DPS. Penalty is defined as a combination of functions:

- Health function,

$$hp_p = \begin{cases} (hp - hp_c) \cdot hp_{pm}, & \text{if } hp < hp_c, \\ \frac{hp - hp_c}{4} \cdot hp_{am}, & \text{if } hp \geq hp_c, \end{cases}$$

  where hp is HP, hp_c is HP constraint, hp_{pm} is HP penalty multiplier, and hp_{am} is HP award multiplier.

- Movement speed function,

$$ms_p = \begin{cases} -(ms_c - ms)^{1.5} \cdot ms_{pm}, & \text{if } ms < ms_c, \\ (ms - ms_c)^{0.77} \cdot 2 \cdot ms_{am}, & \text{if } ms \geq ms_c, \end{cases}$$

  where ms is movement speed, ms_c is movement speed constraint, ms_{pm} is movement speed penalty multiplier and ms_{am} is movement speed award multiplier.

- HP regeneration function,

$$hpr_p = \begin{cases} -(hpr_c - hpr)^{1.75} \cdot 3 \cdot hpr_{pm}, & \text{if } hpr < hpr_c, \\ (hpr - hpr_c)^{0.9} \cdot 3 \cdot hpr_{am}, & \text{if } hpr \geq hpr_c, \end{cases}$$

  where hpr is HP regeneration, hpr_c is HP regeneration constraint, hpr_{pm} is HP regeneration penalty multiplier, and hpr_{am} is HP regeneration award multiplier.

- Penalty,

$$penalty = (-hp_p - ms_p - hpr_p) \cdot \frac{dps}{avgdps}$$

$$\text{EDPS} = dps - penalty$$

  where dps is the specimen's DPS and $avgdps$ is average DPS globally. Average DPS is user-defined (default is 1500) and it helps the penalty scale with both big and small DPS numbers.

Penalty and award multipliers are user-defined variables that can be changed in configuration files. With these functions genetic algorithm will prioritize those specimens that fit the constraints. However, if a specimen could gain a lot of DPS to offset the penalty then the constraints will not be met completely. The key here was finding the right numbers so that EDPS was close to DPS unless the specimen deviated too far from the specified constraints.

2.2.10 Genetic Algorithm Parameters

The program initializes itself with parameters supplied from different configuration files. Some of these parameters have already been mentioned, but in order to present a clearer picture they will be defined as follows:

- *ga_verbose* = 1 (default value) — this parameter does not have any impact on functionality of the algorithm. However, it changes the amount of logging that the program outputs. Possible values are 0, 1, 2, 3, with 0 being minimal logging and 3 being heavy logging.

- *ga_number_of_countries* = 4 — number of disjoint sets of populations which also determines the number of concurrent computing threads. The bigger this number, the better the quality of the overall extreme. The minimum number of countries is set to 4.

- *ga_number_of_generations* = 200 — the number of generations this algorithm will run. Once it reaches this number of generations, it will stop.

- *ga_size_of_population* = 300 — the number of specimens representing a population per island. The bigger this number, the better the quality of the local extreme.

- *ga_population_dupes* = 1 — a flag which indicates whether to allow duplicate specimens in a population.

- *ga_tournament_dupes* = 1 — a flag which indicates whether to allow the same specimen to win in multiple tournaments.
- *ga_size_of_tournament* = 8 — the number of specimens that enter the same tournament.
- *ga_size_of_parents* = 0.6 — the percentage of population that is to be recombined into a new population.
- *ga_percentage_elite* = 0.02 — the percentage of population that will be transferred to the next generation untouched. Only the best are selected.
- *ga_percentage_children* = 0.8 — the percentage of population that will be created using crossover method. This indirectly controls the amount of mutations per population. $nMutations = (1 - ga\_percentage\_elite - ga\_percentage\_children) \cdot ga\_size\_of\_population$.
- *ga_percentage_chromosome_crossover* = 0.5 — the percentage of genes that will be exchanged between chromosomes (on average).
- *ga_percentage_chromosome_crossover_substats* = 0.5 — same as above except only for sub-attribute chromosomes.
- *ga_tournament_winner_probability* = 0.65 — the probability for the best specimen to win in a tournament selection process.
- *ga_number_of_mutations_per_gene* = 0.2 — the percentage of genes that will be mutated in each chromosome (on average).
- *ga_depth_search_period* = 9 — the period for performing metaheuristic methods in generation count.
- *ga_multiple_parents* = 1 — a flag which indicates whether a child can have a different second parent for each chromosome (talent tree, sub-attributes and item set).

- *ga_dynamic* = 1 — a flag which indicates whether the algorithm will try to increase the number of mutations and tweak *ga_tournament_winner_probability* when it gets stuck in a local maximum.

- *ga_discard_populations* = 1 — a flag which indicates whether the algorithm will discard its populations when it gets stuck in a local maximum.

- *ga_find_substats* = 1 — a flag which indicates whether it will try to find the optimal combination of sub-attributes using evolutionary methods. If set to false, it will use sub-attributes defined in the hero's configuration file.

- *ga_find_items* = 1 — a flag which indicates whether it will try to find the optimal combination of items using evolutionary methods. If set to false, it will use items defined in the hero's configuration file.

- *ga_permutate_talents* = 1 — a flag which indicates whether it will permute talents when performing metaheuristic methods.

- *ga_permutate_substats* = 1 — a flag which indicates whether it will try to optimize sub-attributes as explained in subsection 2.2.4 when performing metaheuristic methods.

- *ga_permutate_items* = 0 — a flag which indicates whether it will permute items when performing metaheuristic methods.

- *ga_permutate_substats_faster* = 1 — a flag which indicates whether it will perform sub-attribute optimizations independently of other optimizations increasingly faster.

- *ga_use_bias_talents* = 1 — a flag which indicates whether it will remember good talent choices from previous populations when creating new ones.

- *ga_use_bias_items* = 1 — a flag which indicates whether it will remember good item choices from previous populations when creating new ones. It also indicates whether it will enter *exhaustive testing phase*.

- *ga_use_bias_substats* = 1 — a flag which indicates whether it will remember good sub-attribute choices from previous populations when creating new ones.

In Chapter 4, the results will be based on these parameters.

2.3 Parallelization

Genetic algorithms are very easy to parallelize because each specimen in a population can be evaluated independently. The whole program is run in $N + 1$ threads, where N is the number of population islands (recall that the number of islands is at least 4), which is defined by parameter *ga_number_of_countries*. Each population is simulated in its own thread, with its own random number generator, which is based on Mersenne Twister 19937 [10] 32 or 64 bit. The extra thread is used to collect information from the other threads and output it to the console. Randomness is assured by having a main random number generator generate seed values for new random number generators that get assigned to threads.

When creating populations the program creates as many extra processes as there are populations to display each population's information independently. They communicate with each other using named pipes. A named pipe is an extension of the traditional pipe concept on Unix and Unix-like systems, and is one of the methods of inter-process communication [11].

Main thread keeps child threads in check and constantly polls them for information which is then displayed in the main window.

Chapter 3

Warcraft III - The Kingdom of Kaliron Simulation

This chapter will explain how Warcraft III as well as The Kingdom of Kaliron map work technically. All the details have to be implemented in the simulation for accurate results. We assume that the game is played over internet or on a local network because it is a multiplayer map.

3.1 Mechanics

Game mechanics are rule based systems/simulations that facilitate and encourage a user to explore and learn the properties of their possibility space through the use of feedback mechanisms [12].

3.1.1 Animations

Whenever an action is performed, it is accompanied by an animation. This is done for two reasons: it makes the game more interactive and masks the ping in multiplayer sessions.

These animations delay actions by their duration. Every animation has an event that triggers at some point during the playback which executes the action. They can be cancelled at any time but this may cancel the

action if done incorrectly and the action would have to be restarted. Proper management of cancelling animations can lead to a significant increase in DPS.

3.1.2 Control and Commands

Each player controls one hero from a top down view. Players can issue commands using user interface or keyboard shortcuts:

- *Move* — to move our hero we have to right click anywhere on the terrain and it will proceed to move there.
- *Automatic attack* — this command is executed by right clicking on an enemy when we have our hero selected. The hero will enter an attacking mode where it will try to perform basic attacks as often as possible until interrupted.
- *Use ability* — to use an ability, we have to click on an icon in user interface or use its keyboard shortcut.
- *Use consumable* — same as above.

Whenever a command is issued it takes some time to be executed depending on the player's ping to the server and an internal Warcraft III delay because it uses deterministic lockstep network synchronization [13].

3.1.3 Units

Units in Warcraft III are defined as 3D movable objects that can perform actions. They also have at least these attributes:

- *HP* — when this reaches zero, the unit dies,
- *mana* — required for using abilities,
- *movement speed* — how fast it can move across the terrain in units per second, the default value is 300,

- *armor* — reduces damage taken from physical attacks,

- *damage* — determines how many points it will subtract from the target's HP when it is hit with a basic attack.

For the purposes of this thesis, every unit (when ordered) can automatically attack either in melee or from range and cannot die.

3.1.4 Damage Types

Warcraft III shipped with several damage types [14]. However, in The Kingdom of Kaliron this has been simplified to just two damage types: *physical* and *magical*.

Magical damage is not reduced by the target's armor and will always do full damage. This type often belongs to abilities of heroes attacking at range.

Physical damage is reduced by the target's armor according to formula (3.1). This type belongs to all basic attacks and most of the abilities of heroes attacking in melee.

3.1.5 Critical Hits

Critical hits are attacks that hit a critical part of the enemy, thus dealing significantly more damage. In The Kingdom of Kaliron they can happen as a random chance on each attack. Both physical and magical damage types have their own critical hit chances and multipliers which can be increased with items, talents and sub-attributes.

Since this mechanic introduces randomness into the simulation, it had to be converted into a deterministic behaviour. Let there be critical hit chance p and critical hit multiplier m, then the actual damage is calculated as $d' = d \cdot (1 + p \cdot m)$ where d is the damage an attack would do without a critical hit.

3.1.6 Basic Attack

Basic attack, also called *auto attack*, is a projectile that deals damage when it hits its target as shown in Figure 3.1. It cannot be evaded, always deals physical damage, and it can cause a critical hit. Basic attacks also have a maximum range at which they can be fired. For melee basic attacks the unit has to stand next to the target, whereas for ranged basic attacks they can fire from a specified range.

The sequence of every basic attack can be described as: *begin attack animation*→*attack point* (deals damage)→*back to idle animation*. Interrupting *back to idle animation* to use abilities or begin a new attack yields more DPS.

3.1.7 Armor

Armor is an attribute that determines how resistant a unit is to physical attacks. It does nothing to protect the unit from magical attacks. The formula for physical damage calculation is:

$$f(damage) = damage \cdot (1 - \frac{armor \cdot 0.01}{1 + armor \cdot 0.01}) \tag{3.1}$$

3.1.8 Heroes

Heroes are units that can gain experience by killing other units. They can also use abilities and consumables.

3.1.9 Abilities

Abilities or spells are special actions that a unit can perform if it has enough mana to spend. Once used, they enter a cooldown state during which they cannot be used again. Each ability has its own mana cost and cooldown period. Abilities are limited by their range just like basic attacks.

Abilities can affect enemy units, the hero itself, allied units, or a combination of those. They can be targeted, meaning that once they are selected you have to point and click somewhere, or instantly used. Effects of abilities

can be damaging, healing, empowering or debilitating. Abilities can also be *channeling*, which indicates that once used, the hero will have to wait for the channeling to complete or cancel it before being able to do anything else.

The sequence of abilities that are not instantly used is like that of basic attacks: *begin ability animation*→*cast point* (launches projectile, creates an effect, ...)→*channeling* (if applicable).

Buffs

Buffs are effects given by abilities to units that boost certain aspects of those units for a duration. For example there could be an ability that increased mana regeneration of a unit by ten per second for the next thirty seconds.

Debuffs

Debuffs are effects given by abilities to units that weaken certain aspects of those units for a duration. For example there could be an ability that decreased armor of a unit by twenty for thirty seconds.

3.1.10 Items

Items are objects that heroes can equip using the inventory system in The Kingdom of Kaliron as shown in Figure 1.3.

3.1.11 Consumables

Consumables are objects that are held by heroes up to a maximum of ten consumables. They can be used just like abilities, however, they do not cost any mana to use. After a type of consumable is used, none of the consumables of that type can be used for the duration of its cooldown period.
Types of the most common consumables are:

- *Mana potion* — they come in small, medium, large and huge sizes. When used, it restores a certain amount of mana to the hero.

- *Health potion* — they come in small, medium, large and huge sizes. When used, it restores a certain amount of HP to the hero.

- *Mana ward* — When used, target an area on the ground to place this ward. The ward will regenerate 2.5 percent of mana every three seconds for thirty seconds for every hero in its range.

3.2 Setup

The basis of this simulation is a player attacking a practice dummy for a user-defined length of time (*fight_duration* variable in configuration file). During this time the player would constantly evaluate possible actions and choose the best one available. This would maximize the damage output among other things. After the specified length of time, all the damage done is added to one number which is then divided by the time it took to cause that damage.

Practice dummy is a unit situated in a town (Figure 3.1) that stands still and regenerates life so that it can never die. Armor on this unit can be changed with an in-game command. Because it is in a town, there is no external interference and we are free to focus only on doing as much damage as possible.

The simulation also has a few different parameters that simulates dodging projectiles, avoiding ground effects, and running from enemy units. This allows us to approximate complex fights that cannot be tested in-game on a practice dummy.

3.3 AI Player

Since there is no human interaction during simulation, AI had to be created that would mimic human actions. The AI's decision making skills are approximately as good as the best player's with default configuration but they can be made better or worse.

Figure 3.1: Basic setup of the simulation. A hero attacking practice dummy. The projectile in the air is the hero's basic attack.

AI simulates human perception of ability cooldowns by delaying ability selection for a specified amount of milliseconds (*ability_humandelay* in configuration file). It also simulates misclicks (i.e. selecting wrong ability sometimes) by applying a delay before using the next ability. There are a few more human-like factors that will be explained in the next section. Without the ability to behave like a human AI was outperforming in-game results.

AI evaluates all the possible actions every millisecond as depicted in Algorithm 3.2. Warcraft III does not run at 1000 frames per second (FPS) so this could be changed to reduce running time in the future.

Algorithm 3.2 AI Evaluation Function

```
1:  t ← 0
2:  for t < fightDuration do          ▷ fightDuration is in milliseconds
3:      ExecuteMovement               ▷ performs movement if it is time to
                                        avoid projectiles
4:      PerformAutoAttack
5:      SelectTheBestAbilityToCast
6:      RegenerateMana
7:      i ← 0
8:      for i < nAbilities do
9:          Update(abilities[i])      ▷ updates i^th ability (cooldown, vari-
                                        ous delays, ...) and uses it if se-
                                        lected and ready
10:     Update(target)                ▷ updates debuff durations on the tar-
                                        get we are attacking
```

3.3.1 Movement

To simulate more complex fights, the simulation AI supports artificial movement. In real fights, this usually happens when we have to dodge flying projectiles or hazardous ground effects. Heroes cannot perform any basic attacks or use abilities during movement so such fights will drastically reduce

the potential DPS.

There are several presets of movement available in the configuration file ranging from *none* (standing still) to *heavy* (move 800 units every ten seconds). There is also dodging which works similarly but rather than moving 800 units at once, move several times for 120 units. Abilities differentiate between movement and dodging phases; abilities cannot be used during movement phase, however, they can be used in dodging phase if they are abilities that can be cast from range.

3.3.2 Choosing Abilities

The way AI chooses abilities is defined by each ability's damage per execution time (DPET).

$$\text{DPET} = \frac{d_a}{t_{channel} + t_{cd} + t_{lag} + t_{click} + t_{anim} + t_{movement}}$$

where d_a is damage of the ability, $t_{channel}$ is channeling time left, t_{cd} is cooldown time left, t_{lag} is lag time left (due to ping), t_{click} is target clicking time left, t_{anim} is animation time left, and $t_{movement}$ is movement time left (only applicable in dodging phase). If there is no ability currently selected to be used, AI will iterate through all abilities and calculate DPET for them and pick the one with the highest DPET and lock it in. This way, it will always do the maximum amount of damage possible.

Between using abilities, AI will try to perform basic attacks. Similar to abilities, they also have DPET and can be interrupted if an ability with higher DPET comes off cooldown and vice versa.

3.4 Configuration Files

There are several configuration files. The main configuration file is named *config.cfg* which is responsible for general simulation and genetic algorithm parameters. Beside those, there is at least one configuration file for every type of hero (e.g. *arc1.cfg*), which defines variables such as hero level, any

buffs or debuffs it might have, possible item choices[1], sub-attribute point assignments[2], and game related parameters like sub-attribute gains per level-up.

3.4.1 Simulation Parameters

Simulation parameters are located in a configuration file named *config.cfg* next to the executable file. These are defined as follows:

- *hero_config* — tells the program which hero configuration file to use.
- *fight_duration* = 300 (default value) — the length of the fight in seconds.
- *fight_target_armor* = 33 — the amount of armor on the target.
- *fight_chainmanaward* = 0 — whether the mana ward consumable will permanently be up (1) or not (0).
- *fight_unlimitedmana* = 0 — whether the hero has unlimited mana (1) or not (0).
- *fight_movement* = medium — the amount of movement in the fight. Possible options are none, light, medium and heavy.
- *fight_dodge* = medium — the amount of projectile avoidance in the fight. Possible options are none, light, medium and heavy.
- *fight_aura_mps* = 0 — the amount of mana regeneration per second in the fight. Some specific fights can have a negative impact on heroes' mana regeneration.

[1] Item choices limit genetic algorithm's search space. These can be specified if the user knows what they want to equip the hero with.

[2] If genetic algorithm will not be performing a sub-attribute search (defined by parameter *ga_find_substats*), these will be ignored.

- *ability_lag* = 150 — the amount of time in milliseconds after queuing up an action before it is performed. This is due to the client's ping to the server (average).
- *ability_castdelay* = 150 — the amount of time in milliseconds it takes a player to react to the end of an ability's animation (average).
- *ability_clicktime* = 60 — the amount of time in milliseconds it takes a player to issue the target on targetable abilities (average).
- *ability_humandelay* = 150 — the amount of time in milliseconds it takes a player to notice and select an ability when it comes off cooldown (average).
- *ability_errorrate* = 4 — number of actions out of 100 that a player will err on.

By changing above parameters we can approximate any kind of fight in the simulation – from standing still and attacking to moving constantly and having very little time to attack. We can even experiment and change internal numbers to simulate different scenarios to find ways to balance the game.

Chapter 4

Results

4.1 Testing Environment

Before presenting the results, it is important to describe the testing environment used for running the program. The computer used for testing was a Windows 7 x64 machine with an Intel© Core™ i5-2500K Quad-Core CPU. The program was compiled with Microsoft Visual Studio 2013 in release mode with optimizations enabled.

The algorithm was implemented in C++ programming language (version C++11) using OpenMP API for multithreading support.

4.2 Genetic Algorithm Performance Comparison

In this section we will compare performances of a typical genetic algorithm and our revised genetic algorithm. In Figure 4.1 we observe the development of our genetic algorithm. There are significant improvements in the early generations, however, very little improvement is seen later. This is very characteristic of genetic algorithms, however, the point at which they reach maximum solution quality differs.

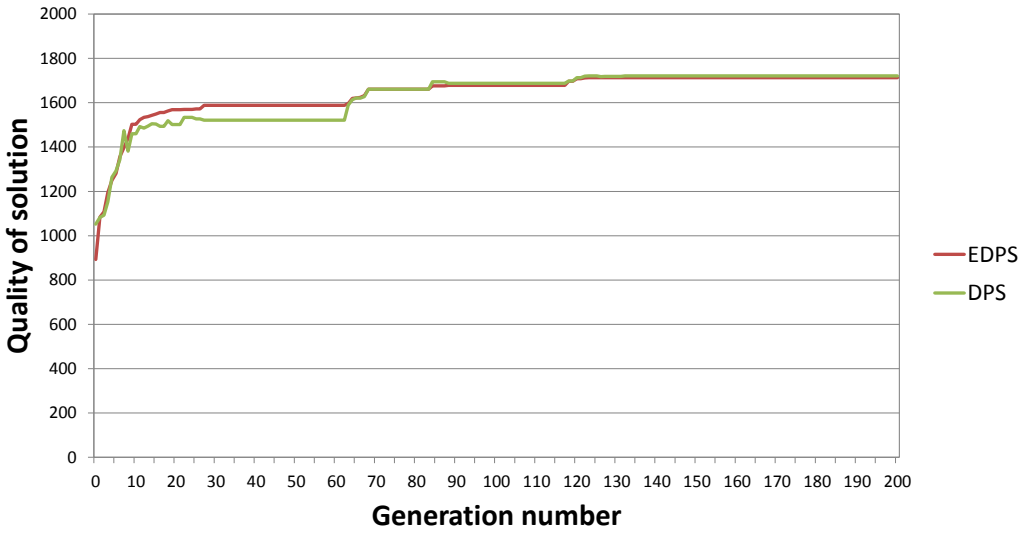

Figure 4.1: DPS and EDPS over the course of our genetic algorithm execution. We do not show the effect of discarding populations here.

In the following subsections the parameters used are described in sections 2.2.10, 3.4.1 (unless otherwise noted) with *hero_config* set to *rang1_proper.cfg*.

4.2.1 Typical Genetic Algorithm

We had to disable all our improvements (Table 4.1) to see how a typical genetic algorithm would perform. We observe (Table 4.2) that it performs much worse than our revised genetic algorithm (Table 4.4) even with greater population size (Table 4.3). It appears that the genetic algorithm without our improvements barely makes any progress beyond generation 50. This is due to the fact that it has no mechanisms that could redirect it to a different search space (e.g. population discarding). The results at the end also have a high standard deviation, meaning the final results are often suboptimal.

Table 4.3 is a good example of how a greater population size does not help the genetic algorithm avoid local maxima. The results compared to Table 4.2 are better overall; some of them even reached a solution quality

parameter	value
ga_number_of_countries	1
ga_depth_search_period	0
ga_multiple_parents	0
ga_dynamic	0
ga_use_bias_talents	0
ga_use_bias_items	0
ga_use_bias_substats	0

Table 4.1: Non-default parameter choices. Used in a typical genetic algorithm run, without our improvements.

generation	Avg. EDPS ± std. dev.
10	1433.48 ± 44.4369
25	1562.02 ± 48.0540
50	1590.30 ± 55.7566
100	1604.16 ± 45.2645
200	1605.82 ± 45.7344

Table 4.2: Performance of a typical genetic algorithm at *ga_population_size* = 300.

similar to that of revised genetic algorithm but they did not find the best solution due to the lack of metaheuristic methods to optimize specimens near global maximum.

4.2.2 Revised Genetic Algorithm

Our revised genetic algorithm performed very well. It did not always reach the optimal solution[1], however, the results were consistent and satisfactory considering the complexity of the problem.

According to the results (Table 4.4), our implementation of genetic algorithm converges towards a global maximum slowly and it does not get stuck

[1]We do not know with certainty that it ever reached the exact optimal solution.

generation	**Avg. EDPS ± std. dev.**
10	1491.82 ± 31.9751
25	1613.94 ± 40.2547
50	1631.88 ± 47.0037
100	1635.27 ± 47.6178
200	1648.03 ± 55.4460

Table 4.3: Performance of a typical genetic algorithm at *ga_population_size* = 1200.

generation	**Avg. EDPS ± std. dev.**
10	1567.08 ± 34.3537
25	1656.05 ± 42.5592
50	1662.92 ± 36.7452
100	1687.61 ± 17.6048
200	1710.06 ± 3.06402

Table 4.4: Performance of our revised genetic algorithm at *ga_population_size* = 300.

when hitting a local maximum like the basic genetic algorithm. Solving this, while keeping run times within acceptable margins, was the most difficult challenge during the development of our algorithm.

4.3 Influence of Parallelization

The implementation of the algorithm parallelizes execution by running each population in its own thread. For that reason it was not straightforward to compare execution speed between one or more cores. To simulate the same workload on fewer cores (meaning fewer populations) we had to increase population size. The result is not entirely the same but it is a very close approximation.

The following parameters were chosen for different amounts of threads:

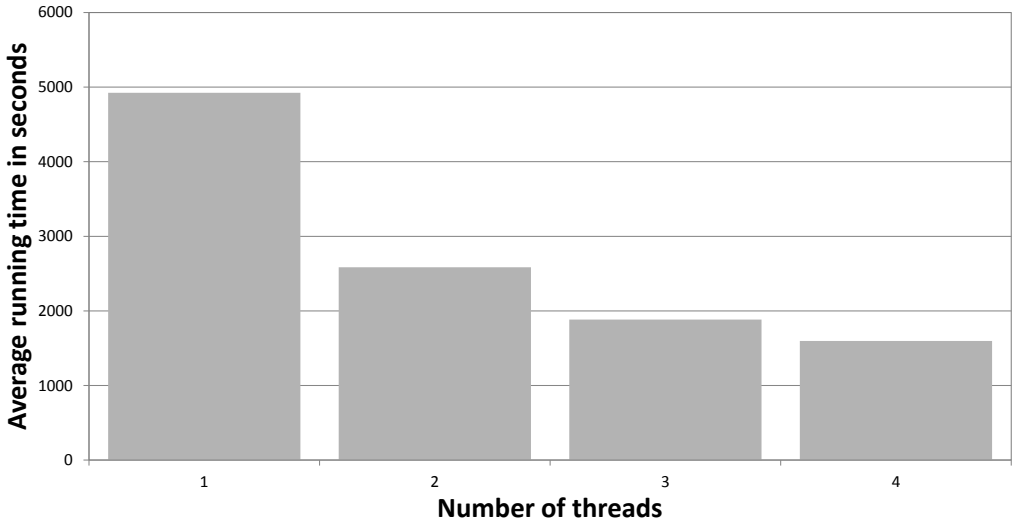

Figure 4.2: Influence of parallelization.

- 1 thread: $ga\_number\_of\_countries = 1$, $ga\_size\_of\_population = 1200$,
- 2 threads: $ga\_number\_of\_countries = 2$, $ga\_size\_of\_population = 600$,
- 3 threads: $ga\_number\_of\_countries = 3$, $ga\_size\_of\_population = 400$,
- 4 threads: $ga\_number\_of\_countries = 4$, $ga\_size\_of\_population = 300$.

Note that one of the threads is idle for the most part (see Section 2.3) so we excluded it in this comparison.

From Figure 4.2 we can see that they scale nearly linearly. The reason why four threads were not much faster than three is most likely due to the operating system's background tasks that consumed some of the CPU during processing.

4.4 Hero Comparison

In this section we will present DPS numbers for different heroes at different levels to see how they compare. Such a comparison will give us insight into

hero type	1	2	3	7	10	15	20	25	30	35	40	60
Arcanist	12.74	24.29	26.52	47.86	91.91	159.20	266.72	478.62	1042.35	1515.98	1762.39	2889.61
Barbarian	14.18	25.25	29.06	59.44	106.99	170.98	266.31	485.97	1011.19	1369.93	1603.77	2405.94
Druid Caster	15.70	21.96	26.90	57.96	113.17	202.34	307.74	517.36	1155.98	1619.20	1823.95	2733.01
Druid Wolf	1.79	12.80	16.05	46.95	101.66	179.27	313.60	543.03	1168.48	1585.19	1802.78	2585.12
Phantom Stalker	15.03	21.65	26.62	56.34	120.66	221.26	393.91	809.18	1522.68	1942.64	2273.34	3295.77
Pyromancer	12.43	15.72	18.76	36.70	79.68	143.89	276.14	507.30	1039.47	1535.30	1859.46	2900.91
Ranger	17.28	28.72	32.23	65.10	115.35	193.96	316.62	585.55	1239.58	1756.46	1996.32	3094.85

Table 4.5: A DPS per level comparison of different hero types across the whole level spectrum. The only constraint applied was HP regeneration being above -15.

how heroes develop as they gain levels. Parameters used were described in sections 2.2.10, 3.4.1.

4.4.1 Hero Comparison Throughout the Game

In The Kingdom of Kaliron maximum level of a hero is defined as 60. However, due to the unfinished nature of the map, maximum level one can achieve is actually 40.

As we start playing the game, we all start at level 1. It would be interesting to know how different hero types fare throughout the game, potentially finding flaws in item or hero design.

With our program, we can simulate level 60 with currently available items and talents. As we can see from Table 4.5, heroes seem to be almost equivalent to each other at around level 40. They are no longer balanced at level 60 because the map developers have not created items for levels past 40 yet. Items play a major role in determining a hero's DPS as we can see in the example of a Pyromancer. Pyromancer does very little DPS at low levels because there are no items at those levels, however, it picks up after level 25.

The discrepancy in DPS numbers is also due to the roles different hero types play in fights. For example, a Barbarian gives all their allies a HP regeneration buff and all enemies receive an armor debuff, while a Phantom Stalker is only meant to do as much damage as possible. This greatly influences item choices of other heroes in their party, potentially increasing their

hero type	DPS at level 40
Arcanist	1533.46
Barbarian	1555.42
Druid Caster	1570.55
Druid Wolf	1565.70
Phantom Stalker	1759.44
Pyromancer	1705.97
Ranger	1721.07

Table 4.6: A DPS comparison of different hero types at maximum level 40.

DPS by a significant margin.

4.4.2 Hero Comparison at Maximum Level

Once heroes reach maximum level in The Kingdom of Kaliron, they are left with to strive for items only. Item collection is a very important gameplay experience in The Kingdom of Kaliron. The best items are found by defeating bosses[2], so assembling the optimal party for a specific boss is important. In Table 4.6 we compare how well different types of heroes do at maximum level (40). They were all setup with proper HP, HP regeneration, and movement speed constraints.

From Table 4.6 we can see that the pure damage dealers (Ranger, Phantom Stalker, Pyromancer) are doing the most DPS while the others are rather equal amongst themselves. The Kingdom of Kaliron appears to be pretty balanced at maximum level when we take into account the necessary attributes we require to participate in those fights.

[2]Bosses are very difficult fights in The Kingdom of Kaliron.

4.5 Speed or Quality

During the testing of the algorithm, we noticed that it did not produce the same results every time. This was not unexpected and it could be corrected by adjusting genetic algorithm parameters to allow for bigger populations and more population islands. However, we settled for the configuration that allowed us to get results within 20 minutes, which were often the optimal. In case they were not, they would not deviate from the optimal results for more than 0.5 percent (Table 4.4).

Chapter 5

Conclusions and Future Work

The purpose of this thesis was to find the optimal solution for a combinatorial problem where a brute force approach was not feasible.

It would save a lot of work if there was some kind of template for creating a genetic algorithm, but it seems that there is not. Each problem has its own set of parameters and heuristics that can be applied during the development of a genetic algorithm. It is up to the developer to figure out the best heuristics and sets of parameters and their values.

In Section 4.4 we have shown that the quality of the solution, considering the running time, is very good. However, there is no way for us to know if a solution is the global maximum because there is no algorithm that would solve this problem in a reasonable time.

Tweaking and selecting of genetic algorithm parameters and development of various heuristics was mostly done through trial and error and a lot of experimentation. In Section 4.2 we show that the algorithm without our heuristics performs much worse.

Using this program, authors of the map The Kingdom of Kaliron can tweak heroes' attributes and ability effects to find a state where each hero performs similarly to others. This is called *balance*. The program is an example of a tool that could be helpful during game development for any kind of game.

For future work in this field there are a few things that could be improved or changed:

- The genetic algorithm depends on several user-defined parameters. Instead of manually tweaking these parameters, we could use another genetic algorithm to determine the optimal configuration. However, this configuration would only be valid until a new hero was added. That is due to the fact that genetic algorithms tend to adapt to the situation — with an additional hero it would have to relearn the optimal configuration.

- We could implement a new deterministic search method which would be run after the genetic algorithm has finished processing. This search method would deterministically try to find the optimal solution on specimens near global maximum.

Bibliography

[1] Wikipedia, "Warcraft iii: Reign of chaos — wikipedia, the free encyclopedia," 2015, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Warcraft_III:_Reign_of_Chaos&oldid=696200011

[2] ——, "Dota 2 — wikipedia, the free encyclopedia," 2016, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Dota_2&oldid=698087080

[3] ——, "Genetic algorithm — wikipedia, the free encyclopedia," 2016, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=697780185

[4] S. Sarmady, "An investigation on genetic algorithm parameters," [Online; accessed 26-December-2015]. Available: http://sarmady.com/siamak/papers/genetic-algorithm.pdf

[5] "Guide — explanation of W3M and W3X files," [Online; accessed 22-December-2013]. Available: http://world-editor-tutorials.thehelper.net/cat_usersubmit.php?view=42787

[6] Wikipedia, "Jass — wikipedia, the free encyclopedia," 2015, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=JASS&oldid=685291938

[7] "About Cheat Engine," 2015, [Online; accessed 21-December-2015]. Available: http://www.cheatengine.org/aboutce.php

[8] Wikipedia, "Tournament selection — wikipedia, the free encyclopedia," 2015, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Tournament_selection&oldid=676563474

[9] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/937503.937505

[10] Wikipedia, "Mersenne twister — wikipedia, the free encyclopedia," 2016, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Mersenne_Twister&oldid=698152946

[11] ——, "Named pipe — wikipedia, the free encyclopedia," 2015, [Online; accessed 6-January-2016]. Available: https://en.wikipedia.org/w/index.php?title=Named_pipe&oldid=687316985

[12] "Lost garden: What are game mechanics?" 2015, [Online; accessed 6-January-2016]. Available: http://www.lostgarden.com/2006/10/what-are-game-mechanics.html

[13] G. Fiedler, "Gaffer on games | deterministic lockstep," 2015, [Online; accessed 6-January-2016]. Available: http://gafferongames.com/networked-physics/deterministic-lockstep/

[14] "Warcraft iii — basics → armor and weapon types," 2015, [Online; accessed 6-January-2016]. Available: http://classic.battle.net/war3/basics/armorandweapontypes.shtml